

## Homework 9

### Due Wednesday, 6/1/11

Please turn in your written homework solutions, program listings, and test runs before the beginning of class on the due date and put your programming solutions in the dropbox on Blackboard.

#### 1. (10 points) **Atomicity & Race Conditions**

The `DoubleCounter` class defined below has methods `incrementBoth` and `getDifference`. Assume that `DoubleCounter` will be used in multi-threaded applications.

```
class DoubleCounter {
    protected int x = 0, y = 0;

    public int getDifference() {
        return x - y;
    }

    public void incrementBoth() {
        x++;
        y++;
    }
}
```

There is a potential data race between `incrementBoth` and `getDifference` if `getDifference` is called between the increment of `x` and the increment of `y`. For the following questions, assume that the “++” operators are atomic (in reality they are not) and that each thread calls `incrementBoth` exactly once.

- (a) What are the possible return values of `getDifference` if there are 2 threads?
- (b) What are the possible return values of `getDifference` if there are  $n$  threads?
- (c) Data races can be prevented by inserting synchronization primitives. One option is to declare

```
public synchronized int getDifference() {...}
public void incrementBoth() {...}
```

This will prevent two threads from executing method `getDifference` at the same time. Is this enough to ensure that `getDifference` always returns 0? Explain briefly.

- (d) Is the following declaration

```
public int getDifference() {...}
public synchronized int incrementBoth() {...}
```

sufficient to ensure that `getDifference` always returns 0? Explain briefly.

(e) What are the possible values of `getDifference` if the following declarations are used?

```
public synchronized int getDifference() {...}
public synchronized int incrementBoth() {...}
```

2. (10 points) **Concurrent Access to Objects**

Please do problem 14.6 from Mitchell, page 471.

3. (10 points) **Java synchronized objects**

Please do problem 14.7, parts a,b,c, and e, from Mitchell, page 472.

For part e, consider the following variant of the program (rather than the modifications they suggest in part d, which you get to skip):

BoundedBuffer from the text:

```
public class BoundedBuffer {
    protected int MaxBuffSize;

    private int[] buffer;

    private int numSlots = 0;
    private int takeOut = 0, putIn = 0;
    private int count=0;

    private Object lockPut = new Object();
    private Object lockGet = new Object();

    public BoundedBuffer(int numSlots) {
        if(numSlots < 0) {
            throw new IllegalArgumentException("numSlots <= 0");
        }
        this.numSlots = numSlots;
        buffer = new int[numSlots];
    }

    public void put(int value) throws InterruptedException {
        synchronized (lockPut) {
            while (count == numSlots)
                lockPut.wait();
            buffer[putIn] = value;
            putIn = (putIn + 1) % numSlots;
            count++;
        }
        synchronized(lockGet){
            lockGet.notifyAll();
        }
    }
}
```

```

public int get() throws InterruptedException {
    int value;
    synchronized (lockGet) {
        while (count == 0)
            lockGet.wait();
        value = buffer[takeOut];
        takeOut = (takeOut + 1) % numSlots;
        count--;
    }
    synchronized(lockPut){
        lockPut.notifyAll();
    }
    return value;
}
}

```

Note how the use of two different locks allows us to avoid two threads executing the `get` method or two threads executing the `put` method, but allows one thread to execute `get` while another executes `put`. Also notice that Java only allows a method to send a `notifyAll()` message to a lock if it holds the lock. Thus “put” must grab the “lockGet” lock before it can notify those processes waiting to do a `get` that there are now elements that can be grabbed.

#### 4. (20 points) Using Bounded Buffer

The Haskell function, `primesto n`, given below, can be used to calculate the list of all primes less than or equal to `n` by using a variant of the classic “Sieve of Eratosthenes”. In CS 8 we saw that using streams we could use this code to generate a stream of all primes by calling `primes = sieve [2..]`, but because there are no streams in Java we will only do the finite version.

```

{- Sieve of Eratosthenes: Remove all multiples of first element from list,
   then repeat sieving with the tail of the list. If start with list [2..n]
   then will find all primes from 2 up to and including n. -}
sieve :: (Integral a) => [a] -> [a]
sieve [] = []
sieve (fst:rest) = let
    nurest = filter (\n -> (n `mod` fst) /= 0) rest
    in
    fst:(sieve nurest)

-- return list of primes from 2 to n
primesto :: (Integral t) => t -> [t]
primesto n = sieve [2..n];

```

Notice that each time through the sieve we first filter all of the multiples of the first element from the tail of the list, and then perform the sieve on the reduced tail. In an eager language, one

must wait until the entire tail has been filtered before you can start the sieve on the resulting list. In Haskell, it will perform the calculation on demand and only calculate as much as is needed. For example if we displayed only the first element of the result of `primesto 100`, then only that element would be calculated. If we have spare processors, one could use parallelism to have one process start sieving the result before the entire tail had been completely filtered by the original process.

Here is a good way to think of this concurrent algorithm that will use the Java `BoundedBuffer` class defined in the previous problem. The main program should begin by creating a `BoundedBuffer` object (say with 5 slots) and then should successively insert the numbers from 2 to  $n$  (for some fixed  $n$ ) into the `BoundedBuffer` using the `put` method, and finally put in -1 to signal that it is the last element. After the creation of the `BoundedBuffer` object, but before starting to put the numbers into the `BoundedBuffer`, the program should create a `Sieve` object (using the `Sieve` class described below) and pass it the `BoundedBuffer` object (as a parameter to `Sieve`'s constructor). The `Sieve` object should then begin running in a separate thread while the main program inserts the numbers in the buffer.

After the `Sieve` object has been constructed and the `BoundedBuffer` object stored in an instance variable, `in`, its `run` method should get the first item from `in` using the `get` method. If that number is negative then the `run` method should terminate. Otherwise it should print out the number (`System.out.println` is fine) and then create a new `BoundedBuffer` object, `out`. A new `Sieve` should be created with `BoundedBuffer out` and started running in a new thread. Meanwhile the current `Sieve` object should start filtering the elements from the `in` buffer. That is, the `run` method should successively grab numbers from the `in` buffer, checking to see if each is divisible by the first number that was obtained from `in`. If a number is divisible, then it is discarded, otherwise it is put on buffer `out`. This reading and filtering continues until a negative number is read. When the negative number is read then it is put into the `out` buffer and then the `run` method terminates.

If all of this works successfully then the program will eventually have created a total of  $p + 1$  objects of class `Sieve` (all running in separate threads), where  $p$  is the number of primes between 2 and  $n$ . The instances of `Sieve` will be working in a pipeline, using the buffers to pass numbers from one `Sieve` object to the next.

Please write this program in Java using the `BoundedBuffer` class from the previous problem. Each of the buffers used should be able to hold at most 5 items.

5. (15 points) **Bounded Buffer using STM**

We have seen bounded buffers implemented with locks and with synchronized code. In this problem you will implement a bounded buffer using Software Transactional Memory. The buffer will be saved as a Haskell list. Even though there is no upper limit to the length of a list, this code should never allow the list to be longer than the bound given.

Fill in the missing bodies for `put` and `get` in the following code:

```

module BoundedBuffer where

import Control.Concurrent.STM

type BoundedBuffer a = TVar ([a], Int) -- The buffer and its maximum size

createBoundedBuffer :: Int -> STM (BoundedBuffer a)
createBoundedBuffer size = newTVar ([], size)

put :: a -> BoundedBuffer a -> STM ()
put item buffer = do

    *** Fill in body here ***

get :: BoundedBuffer a -> STM a
get buffer = do

    *** Fill in body here ***

```

Test your buffer on the following test code.

```

{- Tests the Bounded Buffer module
   by Scot Drysdale on 5/26/11
-}

import BoundedBuffer
import Control.Concurrent
import Control.Concurrent.STM
import Random

-- Delay for a random period up to maxDelay
randDelay :: Int -> IO ()
randDelay maxDelay = randomRIO (0, maxDelay) >>= threadDelay

producer :: Int -> BoundedBuffer Int -> IO ()
producer 0 buffer = do
    atomically $ do put (-1) buffer
                    put (-1) buffer
    return ()

producer n buffer = do
    atomically $ put n buffer
    randDelay 75
    producer (n-1) buffer

```

```
consumer :: BoundedBuffer Int -> Int -> MVar Int -> IO ()
consumer buffer delay printLock = do
  value <- atomically $ get buffer
  if value < 0
  then return ()
  else do
    randDelay delay
    safePrint value printLock
    consumer buffer delay printLock

safePrint :: (Show a) => a -> MVar Int -> IO ()
safePrint value printLock = do takeMVar printLock; print value; putMVar printLock 0

main :: IO ()
main = do
  printLock <- newMVar 0
  buffer <- atomically $ createBoundedBuffer 5
  forkIO $ consumer buffer 100 printLock
  forkIO $ consumer buffer 500 printLock
  producer 25 buffer
```

Turn in a listing of your BoundedBuffer module, a printout of the output, and an explanation of why the numbers do not get printed in strict decreasing order.